LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-670941

Experiences of Using the OpenMP Accelerator Model to Port DOE Stencil Applications

P. Lin, C. Liao, D. Quinlan, S. Guzik

May 18, 2015

International Workshop on OpenMP
Aachen, Germany
October 1, 2015 through October 2, 2015

Disclaimer

Experiences of Using the OpenMP Accelerator Model to Port DOE Stencil Applications

Pei-Hung Lin[1], Chunhua Liao[1], Daniel J. Quinlan[1], and Stephen Guzik[2]

[1] Lawrence Livermore National Laboratory
[2] Colorado State University

Abstract. The Department Of Energy (DOE) has a wide range of large-scale, parallel scientific applications running on cutting-edge High-Performance Computing (HPC) systems to support its mission and tackle critical science challenges. A recent trend in these HPC systems is to add commodity accelerators, such as Nvidia GPUs and Intel Xeon Phi co-processors, into computer nodes so we can achieve increased performance without exceeding the limited power budget. However, it is well-known in the HPC community that porting existing applications to accelerators is a difficult task given the numerous unique hardware features and the complexity of software.

In this paper, we share our experiences of using the OpenMP Accelerator Model to port two DOE stencil applications to exploit Nvidia GPUs. Introduced as part of the OpenMP 4.0 specification, the OpenMP accelerator model provides a set of directives for users to specify semantics related to accelerators so that compilers and runtime systems can automatically handle repetitive and error-prone accelerator programming tasks, including code transformations, work scheduling, data management, reduction, and so on. Using a prototype compiler implementation based on the ROSE source-to-source compiler framework, we report the problems we encountered during the porting process, our solutions, and the obtained performance. Productivity is also evaluated. Our experiences show that the existing OpenMP Accelerator Model can effectively help programmers leverage accelerators. However, complex data types and non-canonical control structures can pose challenges for programmers to productively apply accelerator directives.

1 Introduction

The Department of Energy (DOE) has a wide range of large-scale, parallel scientific applications to support its mission and tackle critical research and development challenges in multiple science disciplines. Many of these scientific

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported by the National Science Foundations Computer Research Infrastructure program under Award No. CNS-1205708.

applications have a lifespan of multiple decades so it is essential to port them to current mainstream high-performance computing (HPC) systems deployed in DOE in a timely fashion. A recent trend in the HPC systems [2] is to add commodity accelerators, such as Nvidia GPUs and Intel Xeon Phi coprocessors, into computer nodes so we can achieve increased performance within a limited power budget. However, it is well-known in the HPC community that porting existing applications to accelerators is a difficult task given the numerous unique hardware features of accelerators and the complexity of software.

Although low-level programming models, such as CUDA [3] and OpenCL [10], can often help deliver competitive performance for given applications, they are not productive solutions to port large-scale parallel applications due to the required radical and comprehensive changes to the original source code. On the other hand, high-level programming models such as OpenMP 4.0 [14] and OpenACC [5] provide language annotations, in the form of directives and clauses, for users to incrementally specify semantics related to how to port to an accelerator. Compilers and runtime systems then automatically take care of repetitive and error-prone code transformation, thread scheduling, data management, and so on. Therefore, it is more productive for users to use high-level directive-based programming models to test the feasibility and profitability of using accelerators.

The OpenMP Accelerator Model, introduced as part of the OpenMP 4.0 specification, is a representative high-level directive-based programming model aimed to simplify the programming for accelerators. It assumes a computation node as a host device connected with one or more accelerators as target devices. Each device has its own memory space though it is allowed to have shared storage among multiple devices. The execution model is host-centric: a host device "offloads" data and code regions to accelerators for execution, but the accelerators do not initiate communication with hosts. A set of directives, environment variables, and runtime library routines are provided to specify semantics related to managing computation and data between the host and target devices.

In a previous study [12], we created a prototype compiler for the OpenMP Accelerator Model and obtained an early impression of its expressiveness, implementation, and performance. In this paper, we extend our work by applying the model to port two non-trivial DOE scientific applications: lattice-Boltzmann method and Compressible Navies-Stokes equation. Both applications conduct a stencil computation, an important category of scientific computing done in DOE facilities. However, they represent a spectrum of stencil applications by their difference in stencil sizes. Our goal is to discover problems developers may face when using the OpenMP Accelerator Model to port real applications. We also share our solutions to the problems, including suggestions to improvements to the programming model itself. Our contributions include: 1) providing the first study of using the OpenMP Accelerator Model in OpenMP 4.0 to port non-trivial scientific applications, 2) illustrating the obstacles for porting real applications and possible solutions and workarounds, and 3) suggesting improvements, including new language features, of the OpenMP Accelerator Model to increase expressiveness and performance for accelerators,

The remainder of this paper is organized as follows. Section 2 gives an overview of the accelerator support in the OpenMP 4.0 specification. Section 3 describes the two chosen applications. Porting experiences are given in Section 4, including details for baseline performance, performance analysis, and optimized versions. Section 5 summarizes related work. Finally, discussion and future work in Section 6.

2 OpenMP 4.0's Accelerator Support

OpenMP is a representative high-level directive-based programming model originally designed to address shared-memory programming. Starting from OpenMP 4.0, it has a set of language directives and runtime routines aimed at simplifying the programming for accelerators. Collectively, the accelerator support is often called the OpenMP Accelerator Model. The OpenMP accelerator model assumes that a computation node has a host device connected with one or multiple accelerators as target devices (shown in Figure 1). A target device, which can be any logical execution engine defined by an implementation, has threads that behave almost the same as threads on the host device. The OpenMP memory model is extended so that the code region has its own data environment. A device appears to have an independent memory, although it is allowed to share memory among devices.

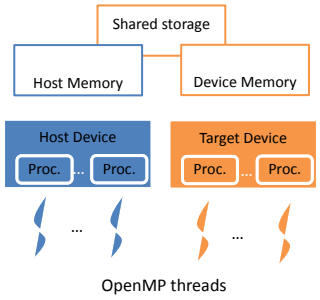


Fig. 1: Devices and memory in OpenMP 4.0

The execution model is host-centric: a host device "offloads" data and code regions to accelerators for execution, but the accelerators do not initiate communication with hosts. In particular, the `target` construct is introduced for specifying a computation and the associated data to be offloaded to a device. Initially, only a single thread starts on a device to run an implicit task region. This single thread can fork more threads later when it encounters parallel constructs. It can also generate tasks, as can its CPU counterpart. Data-mapping attributes, specified using the `map` clause, define how variables are handled for the device data environments, including allocation, initialization and assignment to the host

variables at the end of a device data environment. Data mapping often involves data movement as host and device are commonly in different memory spaces in modern accelerator architectures. To avoid repetitive creation and cancellation of device data environments, the target data directive defines a device data region, in which multiple target regions can share the same device data.

Figure 2 shows a Jacobi iteration kernel written using the OpenMP accelerator model. One directive (line 6 and 7 of Figure 2) converts the existing host OpenMP code to device code. Since a target region can run on a host device whenever an implementation chooses, programmers should generally write a host version before adding accelerator-specific directives.

```
1  #pragma omp target data map(to:n, m, omega, ax, ay, b, f[0:n][0:m]) \
2            map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])
3  while ((k<=mits)&&(error>tol))
4  {
5  // a "target + parallel for" loop copying u to uold is omitted ...
6  #pragma omp target map(to:n, m, omega, ax, ay, b, f[0:n][0:m], \
7                    uold[0:n][0:m]) map(tofrom:u[0:n][0:m])
8  #pragma omp parallel for private(resid,j,i) reduction(+:error)
9  for (i=1;i<(n-1);i++)
10   for (j=1;j<(m-1);j++)
11   {
12     resid = (ax*(uold[i-1][j] + uold[i+1][j])\
13         + ay*(uold[i][j-1] + uold[i][j+1])+ b * uold[i][j] - f[i][j])/b;
14     u[i][j] = uold[i][j] - omega * resid;
15     error = error + resid*resid ;
16   } // the rest code omitted  ...
17 }
```

Fig. 2: Jacobi example

Accelerators are often massively parallel architecture devices that support hundreds or even thousands of concurrent threads with a hierarchical organization. For example, CUDA provides the hierarchy of threads in blocks and grids. OpenMP 4.0 provides the teams and distribute constructs to manage a two-level thread hierarchy. teams creates a league of thread teams, and the master thread of each team executes the region. distribute is closely nested in a teams region to share work among master threads of teams. Other features in the OpenMP accelerator model include a target update directive to make specified items in the device data environment consistent with their original list items, a target declare directive to specify that variables or functions to be mapped to a device, some combined constructs to simplify the programming, and an environment variable (OMP_DEFAULT_DEVICE) to indicate the default device number, and a set of runtime library routines to set and detect information related to accelerators.

3 Applications

In this paper, we chose two stencil applications, one using the lattice-Boltzmann method (LB) and the other solving the compressible Navier-Stokes equation (CNS). Stencil computations are used in many large DOE scientific applications to solve partial differential equations on structured grids. The chosen LB and

CNS algorithms have very different stencil sizes (zero-point vs. 25-point) leading to different computational characteristics. The LB method operates in a *streaming* mode; memory is read once to perform the computation in the zero-point grid site. In the CNS method, memory from a grid site is repeatedly used in all the stencils that include that grid site. Hence, effective caching is extremely important. The performance of the LB algorithm is often limited by bandwidth whereas the performance of the CNS algorithm is often limited by arithmetic resources. These different characteristics can lead to different implementation strategies when porting the applications to a GPU device. We list a high level comparison between two applications in Table 1 and presents details in the following.

Table 1: Comparison between LB and CNS applications

	Language	AMR library	Stencil	components #	# lines in codes
LB	C++	Chombo	0-point	19	4670 (12879 w/ Chombo code)
CNS	Fortran90	BoxLib	9-point in 1D 25-point in 3D	11	1242 (25967 w/ BoxLib code)

3.1 Lattice-Boltzmann Method (LB)

In the lattice-Boltzmann method, hydrodynamics are described by a discrete kinetic equation for a single-particle distribution function [6],

$$\underbrace{f_i(\boldsymbol{j}+\boldsymbol{e}_i\Delta t, t+\Delta t) = \hat{f}_i(\boldsymbol{j},t)}_{\text{Streaming}} \quad = \quad \underbrace{f_i(\boldsymbol{j},t) + \mathcal{L}_{ik}\left(f_k(\boldsymbol{j},t) - f_k^{\text{eq}}(\boldsymbol{j},t)\right)}_{\text{Collision}}. \quad (1)$$

The chosen `LB` application uses Chombo [7], a parallel adaptive mesh refinement (AMR) library used to solve partial differential equations. The domain size selected in the experiment is a 64^3 Cartesian grid structure partitioned into *boxes*, each of size 32^3. A total of 8 boxes cover the problem domain and 8000 time steps are performed in a single experiment. Fig. 3a shows the pseudo code for the `LB` computation. In the experimental setup, a loop in the application iterates over 8 boxes and performs computations to update the grid cells in each box (represented in line 8). Parallelization can be applied to the loop over boxes (line 8) or loops over grid cells (line 11 and line 16). Multi-level parallelization is feasible only if it is supported in the implementation.

3.2 Compressible Navier-Stokes Equations With Constant Viscosity And Thermal Conductivity (CNS)

The `CNS` algorithm is based on finite-difference methods and the equations are:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0, \quad (2)$$

$$\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u}) + \nabla p = \nabla \cdot \boldsymbol{\tau}, \quad (3)$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + p)\boldsymbol{u}] = \nabla \cdot (\lambda \nabla T) + \nabla \cdot (\boldsymbol{\tau} \cdot \boldsymbol{u}), \quad (4)$$

where ρ is the density, $\boldsymbol{u}$ is the velocity, p is the pressure, E is the specific energy density (kinetic energy plus internal energy), $\boldsymbol{\tau}$ is the viscous stress tensor, λ is the thermal conductivity, and T is the temperature.

The problem domain in `CNS` is represented by BoxLib [1], an AMR library very similar to Chombo. As for the LB case, the domain size is 64^3 and partitioned into "Fabs" (Fortran array box), each of size 32^3. 50 time steps are performed and 5 output files are generated during the computation. An outer loop iterates over all available Fabs in the multi-Fab data structure (shown in line 5 in Fig. 3b). Similar to the `LB`, multi-level parallelization is applicable if it is supported in the implementation.

```
1  fi(cells, 19, boxes) = initial data;
2  fiUpdate(cells, 19, boxes) = 0;
3  U(grid, 4, boxes);
4  Macroscopic(U, fi);
5  for (int iTS = 0; iTS != nTimeStep; ++iTS)
6  {
7    int iBox;
8    for (every box)
9    {
10     {  // Advance function
11       for (every cell)
12         Collision(fi, U);
13       Exchange(fi);
14       BC(fi);
15       Stream(fiUpdate, fi);
16       for (every cell)
17         Macroscopic(U, fiUpdate);
18       swap(fi, fiUpdate);
19     }
20   }
21 }
```

(a) LB algorithm pseudo-code

```
1  init_data(U, dx, prob_lo, prob_hi)
2  for (int iTS = 0; iTS != nTimeStep; ++iTS)
3  {
4    int iFab;
5    for (every Fab)
6      {  // Advance function
7        for (1/3 timestep)
8          {
   // Advance 1/3 of timestep in each iter.
9            for (every grid cell)
10             ctoprim(Unew, Q);
11           for (every grid cell)
12             diffterm(Q, D);
13           for (every grid cell)
14             hypterm(Q, F);
15         }
16     }
17 }
```

(b) CNS application pseudo-code

4 Porting to GPUs

In this section, we describe the platform we target, the porting steps, and performance evaluation. Our porting process starts with obtaining baseline performance of OpenMP versions of the applications. Guided by performance analysis, we then incrementally add additional accelerator directives and clauses to show programming effort and performance impact. In particular, we experiment with directives and clauses for data reuse, loop collapsing, loop scheduling and hierarchical thread mapping. We further experiment with a new clause to test the idea of exploiting special caches on GPUs.

4.1 Hardware and Software Configurations

The hardware platform we target has 132 GB of system memory and two 8-core Intel E5-2670 CPUs. The GPU system has two Nvidia K20X GPUs with 2688 GPU cores and 3.5 CUDA compute compatibility. We use a prototype implementation of the OpenMP Accelerator Model, HOMP (Heterogeneous OpenMP) [12], which is built on the ROSE [15] source-to-source compiler

infrastructure developed at Lawrence Livermore National Laboratory. Leveraging ROSE's flexibility to experiment with new language extensions, HOMP adds the OpenMP accelerator support [12], including parsing and code transformations for target, target data, map and so on. HOMP generates CUDA code for the growing demands in GPU programming. The original OpenMP runtime library (referred to as XOMP) for ROSE has been extended to support thread configuration, loop scheduling, data management, reduction and many other required operations on GPUs. The GNU Compiler Collection (gcc-4.4.6) was used to compile the software with optimization flags -O3 and -fopenmp. Nvidia 6.0 SDK and nvcc compiler are used for the CUDA support on the GPU platform. The Nvidia Visual Profiler [4] is used to conduct performance analysis and guide our optimization strategies.

4.2 Baseline performance on CPU

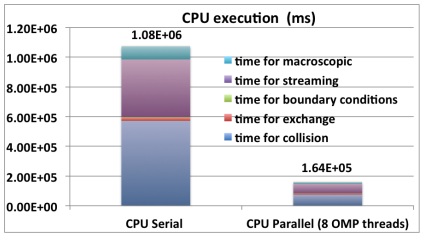
We take the parallel CPU performance as a baseline to compare results from different GPU implementations. The CPU testing platform can support up to 16 hardware threads for the parallel execution. Classic OpenMP directives for CPUs are inserted into the source code to parallelize loop computations.

The default setup in the `LB` application has OpenMP directives inserted into the loop for boxes (line 8 in Fig. 3a). We assign at most 8 OpenMP threads to update the 8 boxes in the loop. Each OpenMP thread will then update 32^3 cells inside a box, a strategy that works well for boxes of this size [13]. The OpenMP parallel region terminates at the end of the loop to from an implicit synchronous barrier between time steps. Fig. 4a shows the CPU's serial and parallel performance. The parallel execution with 8 parallel OpenMP threads delivers a 6.76× speedup compared to the serial execution on Intel E5-2670 CPU.

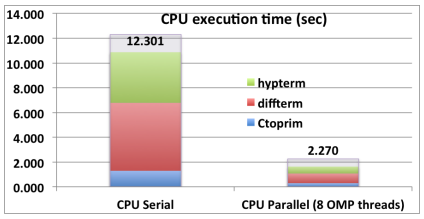
The `CNS` application by default has OpenMP directives at the loops for grid cells (line 9, 11, and 13 in Fig. 3b). The loops in the source code are 3-level nested loops that iterate through the cubical structure in a Fab. The whole application consists 14 such OpenMP parallel loops. In the configured testing case, loop iterations in the outermost loop are evenly distributed into 8 OpenMP threads for 8 boxes. Fig. 4b shows the comparison between serial and parallel execution using 8 threads. The parallel execution delivers 5.42× speedup on the testing machine.

4.3 Baseline performance on GPU

Before performing the porting, we evaluated both applications and discovered a few obstacles to adding OpenMP accelerator directives. We had to modify a subset of code from both applications to make the porting feasible. For example, we used a Fortran-to-C translator implemented in ROSE to translate the computation-intensive functions in the `CNS` into C language versions for the porting. In the `LB` application, several variables used in the target loops are not mappable by the OpenMP 4.0 specification because they are part of other C++



(a) LB baseline performance on CPU



(b) CNS baseline performance on CPU

Fig. 4: CPU baseline performance compared to CPU serial performance

class objects. We copied those variables to temporary independent variables and mapped the temporary variables as a workaround in this study.

The baseline implementations on the GPU simply try to reuse the OpenMP parallel directives without any optimization involved. Minimal OMP target and OMP map directives are used to identify the target region and data to be mapped onto the device.

For the `LB` application, the location of OpenMP directives in the CPU implementation (line 8 in Fig. 3a) is not an ideal start locations for the GPU implementation since it contains multiple kernels in the loop body. Using an incremental approach, we decided to port individual kernels first. We therefore moved the OpenMP directives to the locations of loops to the grid cells inside Collision, Macroscopic and Stream functions (shown at line 11, 15 and 16 in Fig. 3a). These three functions consume the majority of execution time (47% in Collision, 40% in Stream and 7% in Macroscopic) on the parallel CPU execution. The GPU baseline implementation for `CNS` application has OpenMP directives inserted into one loop in the ctoprim function, three loops in the hypterm function, and 7 loops in the diffterm function. Those are the same locations that have OpenMP directives in the parallel CPU implementation. Diffterm function takes the most (34%) portion of total execution in the `CNS` application. Hypterm and ctoprim take 24% and 13% respectively.

The baseline GPU performance in both applications were not competitive compared to their corresponding CPU version performance (shown in Fig. 5 and Fig. 6). After inspection withe the Nvidia Visual Profiler [4], we found that the baseline GPU implementations have extremely low achieved GPU occupancy ($< 2\%$). This is due to the nested loops, identified by the OpenMP directive, which have only small loop iteration sizes in their outermost loop. The translated CUDA codes exploit at most 40 GPU threads to perform the computation and result in low parallelism and performance. The next step in porting is to improve the GPU utilization by increasing the parallelism.

4.4 Increasing Parallelism

Achieving high parallelism is the key for a GPU device to get high computing performance. In addition to optimizing applications for high parallelism, the

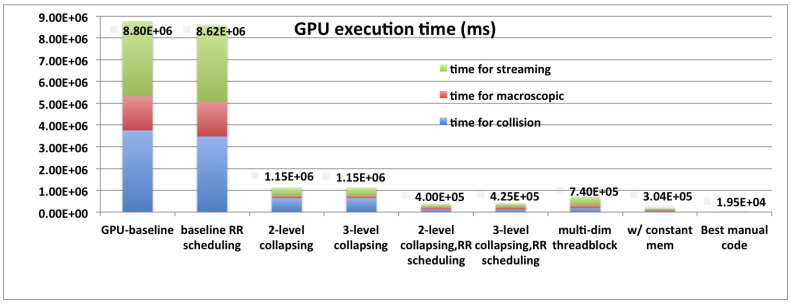


Fig. 5: LB performance on GPU

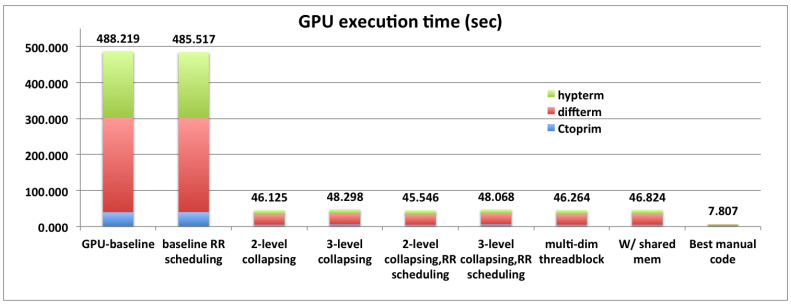


Fig. 6: CNS performance on GPU

porting process needs to take into account that the maximum parallelism in the real execution is subject to certain CUDA limitations. These are the limitations for K20X GPU used in this paper.

- At most 1024 threads in a thread block.
- At most 64 warps (32 threads/warp) in a SMX.
- A thread can have up to a 63 register usage.
- Each SM has up to 48 KB shared memory shared by multiple thread blocks.

There are two feasible approaches to increasing parallelism for the chosen applications. We describe the two approaches with a performance study in this section.

loop collapsing Loop collapsing is a transformation that converts multiple perfectly nested loops into a single loop. Compared to the original outermost loop, the collapsed loop has a larger iteration size with potential to expose higher parallelism. The directive `#pragma omp for collapse (n)` locates the outermost loop and specifies the number of perfectly nested loops to be collapsed. However, loop structure in the `LB` application has statements between the nested loops and does not form a perfectly nested loop. Collapsing non-perfectly nested loops is not allowed by the OpenMP specification. After reviewing the nested loop structure,

we manually moved statements between loops in `LB` application into the innermost loop body since this change causes no side effect and can form a perfectly nested loop. After collapsing, we can have two different parallel executions on 32^2 iterations by collapsing two innermost loops, or 32^3 iterations by collapsing all loops in the example. Therefore, more GPU threads can be assigned to perform parallel execution on the collapsed loop. The XOMP runtime incorporates the CUDA runtime to allocate 16 thread blocks and each has 128 GPU threads. Given 32 registers for each GPU thread, this achieves a 100% theoretical occupancy in each GPU SMX. Collapsing with two-level or three-level loops both can exploit the significant number of threads in a SMX. Compared with the baseline GPU implementations, there are about 5× and 10× speedups delivered for the `LB` and `CNS` applications respectively (shown in Fig. 5 and Fig. 6).

Multidimensional thread structure An alternative option to increase parallelism is using the multidimensional thread structure supported in CUDA so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. The OpenMP 4.0 standard provides the teams directive with optional num_teams() and thread_limit() clauses to support a two-level thread hierarchy. The thread teams and their threads can be mapped to a two-dimensional thread structure in CUDA. In the LB application, we can seamlessly allocate 32×32 threads to update the two innermost loops in the three-level nested loop and have 32 thread blocks mapped to the outermost loop (only two thread blocks can run concurrently due to the limitation). This can achieve 100% occupancy in the execution if only 32 registers are given to each GPU thread. In the `CNS` application, we can have the same allocation if ghost cells are not involved in the computation. Otherwise, the loop iteration size becomes 40 (32 and 4 ghost cells in both sides) in the three-level nested loop. To fulfill the CUDA limitation discussed earlier, we allocate only 40 threads in a thread block and have multiple thread blocks mapped to the loop iteration space (but only up to 16 concurrent thread block executions). This configuration has lower theoretical occupancy (50%) and the computation is inefficient due to usage of partial-warp. The performance is reported in histograms marked with multi-dim threadblock in Fig. 5 and Fig. 6. Compared with the collapsing variants, a 1.5× speedup in overall computation time is achieved in the `LB` application but the `CNS` application showed a minimal difference.

4.5 Loop Scheduling

OpenMP supports multiple loop scheduling policies, including static, dynamic, guided, auto, and runtime. For regular loops running on CPUs, statically and evenly dividing loop iterations among threads using a schedule(static) clause (referred to as static-even schedule in this paper) often leads to the best performance with minimal scheduling overhead. GPUs have much more threads and unique memory access characteristics. On the K20X GPU, every successive 128

bytes (32 single precision words) memory can be accessed by a GPU warp (32 consecutive threads) in a single transaction. The static-even schedule will have one GPU thread accessing multiple successive words in memory and lead to multiple memory transactions. A round-robin scheduling using schedule(static,1) will fulfill the need to perform coalesced memory access on the GPU device. Applying the baseline GPU implementation and two collapsing variants with the round-robin scheduling, performance reports show modest improvement for total execution time in the `CNS` application (1%) and a larger improvement in the the `LB` application (2.8×). Compared only the kernel execution times in the `CNS` application, round-robin scheduling delivers the highest 76% improvement in one kernel in the hypterm function and an average of 26.4% improvement for all kernels. The performance analysis reports high overhead caused by memory movement between the host and device memories. We then take the best-performed variants in both applications (collapsed in two level with round-robin scheduling) to implement optimizations in the following steps.

4.6 Exploiting Memory Hierarchy

Like many other types of accelerators, Nvidia GPUs provide multiple specialized memories, including on-chip software controllable cache shared within a thread block (referred to as shared memory) and constant memory accessible by all threads for read-only global data. The current OpenMP 4.0 lacks support to exploit the specialized memories. We propose to extend the OpenMP Accelerator Model to have a cache clause to allow users to hint such opportunities. The clause has a form of cache (var_list), in which each variable listed can be further prepended by an optional const modifier. For example cache (array1[0:10], const array2[5:10]) tells the compiler that there are two arrays which should be cached in the memory hierarchy of the accelerator. One of the arrays is a read-only subarray. Similar to the map clause, the cache clause can only be used with target or target data directives. Variables shown in the cache clause must also show up in the map clause affecting the same code region. With this clause, compilers translate the code to exploit either shared memory or constant memory of GPUs.

In the `LB` application, there are several values and arrays that can be assigned to the constant memory space on a GPU device. Those include constant coefficients, stride distances (stride over s component or stride over a Fab), an array storing discrete velocity directions and an array storing weights. The `CNS` application has relatively low constant data referenced by multiple functions. Variables passed into the three main computing functions (ctoprim, hypterm, and diffterm) are the data pointers and the upper bound and lower bound of each box structure. Those are only available by retrieving them from the data abstraction and are not suitable to be stored inside the constant memory space. Therefore, no constant memory usage appears in the CNS application. Fig. 7a extracts the comparison (execution time includes memory copying overhead) with only two kernels in the `LB` application to demonstrate the performance with constant memory usage. A 1.32× speedup is achieved for the overall execution time

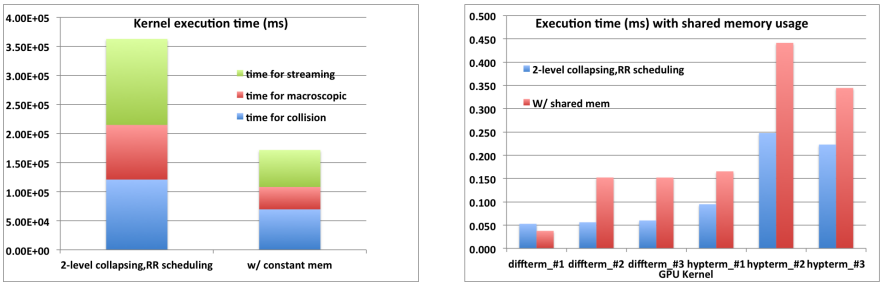
from the implementation with constant memory. Higher speedups, form 1.74× to 2.44×, are observed in the execution times for these three functions individually.

The `LB` application uses a zero-point stencil and the `CNS` application uses a 25-point stencil in the 3D computation, or 9-point stencil if we split computations in three different dimensions. Data represented in a grid cell in the CNS application is referenced multiple times by its neighboring grid cells. A potential to have high performance can be achieved by caching such data in the cache memory, or staged in a temporary buffer with high bandwidth and low latency. The configurable on-chip memory on the GPU device can serve as a L1 cache or shared memory. Stencil data can therefore be stored in the on-chip memory space to gain the benefit of the fast memory. By default, the CUDA programming model tries to cache data fetch into L1 through an implicit memory management. In this experiment, we configure the high capacity in the on-chip memory as a shared memory space to have explicit control of the on-chip memory usage. The primary use of shared memory is to provide inter-thread communication, to cache data to avoid redundant global memory access, and improve global memory access patterns. Multiple threads request the same data element and then generate a significant number of global memory access. To exploit the shared memory space, we can stage the loads in the shared memory space. Only one compulsory global memory access is then required for each data element, the rest of the loads can fetch directly from shared memory with short latency and high bandwidth. However, the implementation needs to pay attention to the memory access pattern to avoid overhead caused by bank conflicts in shared memory. Careless usage of shared memory can also lead to low GPU occupancy and result in low computation performance.

Six kernels (3 in Hypterm and 3 in Diffterm) in the GPU implementation involve the 1D 9-point stencil computation. 40 threads grouped in a thread block will perform computations for grid cells in a 1D array. Data content represented in each grid cell for the Hypterm function has 6 double-precision floating point values and each grid cell in Diffterm function requires 11 double-precision floating point values. Table 2 shows details in required shared memory size, thread assignment and the achievable highest GPU occupancy. This implementation doesn't deliver higher performance compared to our earlier implementation with the best performance (shown in Fig. 7b). The main cause is a much lower GPU occupancy due to low thread number in each thread block. To increase the active thread number in each thread block, loop tiling can be performed to tile multiple iterations in the loop for the second dimension in the 3D nested loop. More threads can be assigned into a thread block but it also proportionally increases the required shared memory size for each thread block. Table 2 shows the changes in GPU occupancy by tiling both kernels with different tiled sizes. With a large tile size, occupancy is limited by the allowed shared memory size (48KB in the configuration). Exploiting shared memory in the `CNS` application does not deliver the best performance. It would require other optimizations to achieve efficient shared memory usage.

Table 2: Shared memory usage and GPU occupancy

Shared memory report			
Kernel	size/block (byte)	threads/ block	Occupancy
Hypterm	1920	40	50%
Tiled 2	3840	80	56%
Tiled 3	5760	120	50%
Tiled 4	7680	120	47%
Diffterm	3520	40	41%
Tiled 2	7040	80	28%
Tiled 3	10620	120	25%
Tiled 4	14080	160	23%



(a) LB performance with constant memory (b) CNS performance with shared memory

Fig. 7: Optimization in memory hierarchy

4.7 Reducing memory movement between host and device

The performance tool reports a significant portion of data copying between the host memory and device memory in all the GPU implementations described above. Several variables and arrays are copied repetitively to the GPU's memory in different kernels. Using target data directives with map clauses can usually reduce repetitive memory allocations and transferring. However, we found that this is not a trivial task for the two chosen applications due to language restrictions.

OpenMP 4.0 defines a set of restrictions for variables listed in the map clause, such as 1) data must have a complete type for C/C++ , 2) a variable that is part of another variable (e.g. a field of a struct) is not allowed unless it is an array element or array section, 3) C++ class types mapped must not contain static data or virtual members, and 4) pointer types are allowed but the memory block to which the pointer refers is not mapped. Chombo (used in `LB` application) and BoxLib (used in `CNS` application) share a data structure called Fortran array box (Fab). Fab is a structure of array that can store multiple components and it provides a high-level data abstraction. Information such as loop bounds, stencil size, and data pointer to the component array are packaged inside the Fab. Members in Fab contain primitive array, scalar variables, and a few static data.

An ideal strategy in the porting process is to copy the entire Fab structure to the GPU's memory space and extract information directly from the Fab stored in GPU memory. However, the Fab structure is not be a mappable according to OpenMP 4.0. A workaround task is extracting and storing all the members in Fab in primitive arrays. Then the temporary arrays can be mapped and copied to the GPU memory. This will involve significant modification in the codes for our chosen applications, and for other scientific applications. We discuss this approach in the paragraph that describes manual optimization.

4.8 Manual tuning for GPU's performance

There are manual implementations for both `LB` and `CNS` applications to evaluate the achievable performance through manual performance tuning. The manual implementations serve as references to study the transformation obstacles in the design of OpenMP accelerator model. Several manual optimizations require good understanding in the application design to perform code modifications and they are not implemented as automatic transformations in this study.

The manual-tuned GPU implementation for the `LB` application was implemented by a coauthor and the tuning process significantly simplifies the Fab structure, restructures the code, and consolidates all the memory copying. Other optimizations include hand-tuned kernels (including BoxLib's exchange function), exploiting constant memory, and several code modifications specifically for GPU implementation. Data is allocated and copied to GPU memory once and reused by all the kernels listed in the pseudo code in Fig. 3a. This optimized implementation delivers the best performance among both the CPU's and GPU's implementations (shown in Fig. 5).

The manual tuning process for the `CNS` application focuses on minimizing memory copying between host and device, exploiting efficient usage of shared memory, and maximizing GPU occupancy without partial warp utilization. A 4^3 thread block is chosen based on the ghost cell size in the computation to avoid the partial warp usage that appears in the implementation with OpenMP accelerator model. This thread block configuration also simplifies the subscript computation when exploiting the shared memory in the execution. The code was modified to have only limited memory transfers between host memory and device memory. All initialized data stored in the Fab data structure is copied to the device memory before the computation. There are infrequent data movements which send only a subset of computed data back to the host memory for boundary exchange performed by BoxLib library and visualization dumps. The manual code delivers the best GPU performance with about $6\times$ speedup compared to the best implementation with OpenMP accelerator model (shown in Fig. 6). However, the delivered performance from GPU is not superior than the performance on CPU. The performance profiling result shows a compulsory overhead in allocating, copying and freeing memory on the GPU. Eliminating those overhead for the `CNS` application, the GPU execution time for the three kernels is at a comparable level to the CPU execution time.

5 Related Work

High-level directive-based programming models, such as OpenMP 4.0 and OpenACC, are becoming promising choices for developers to exploit heterogeneous platforms without compromising productivity. Many previous studies [17,8,11,9] have evaluated the performance and productivity of OpenACC using a range of kernels or applications. For example, Wienke et al. [17] presented their experiences with OpenACC using two real-world applications. OpenACC helped them reach 80% of the best-effort OpenCL version in a moderately complex simulation kernel. They reported that the lack of ability to exploit local memory of GPUs could contribute to the loss of performance of other complex OpenACC applications. Herdman et al. [8] used a hydrodynamics mini-application to compare OpenACC, OpenCL and CUDA. They found that OpenACC was extremely viable. However, their OpenCL and CUDA versions were not optimized. Hoshino et al. [9] used both kernels and a real-world computational fluid dynamics applications to compare CUDA and OpenACC. They reported that some complex Fortran data types such as arrays of derived types and derived types with variable-length arrays are not supported by OpenACC, but extensively used in the code. Manual code changes were used to copy arrays out of complex objects into simpler arrays. Levesque et al. [11] used combination of OpenMP and OpenACC to port S3D. Significant code restructuring efforts were used to raise the level of parallelism, overlap computation and communication, and reduce memory operations.

The application experience of using the OpenMP accelerator support is rare due to the lack of compiler support. Silva et al. [18] compared OpenACC and OpenMP for accelerator computing. A set of parallel programming patterns, not real applications, were used to compare language features. No performance experiments were done due to the lack of compiler support. Unat et al. [16] presented a domain-specific OpenMP-like programming model for stencil methods. For small kernels, they realized up to 80% of the performance of optimized CUDA versions. Our work provides the first study of the performance and programmability of the OpenMP accelerator model using the HOMP compiler [12] and two non-trivial DOE scientific applications. To support software-managed or hardware caches available on accelerators, OpenACC [5] provides a `cache (var_list)` directive to suggest to the OpenACC implementation to fetch the listed variables into the highest level of the cache. However, this directive may only appear inside loops. By contrast, our proposed `cache()` is a clause which can be used with one or multiple code regions (when used with `target data`). Besides leveraging the highest level of cache, the additional `const` modifier in our design can indicate the read-only semantics to exploit constant memory.

6 Discussion & Future Work

Based on our experiments, we have found that the OpenMP Accelerator Model is a productive approach to porting existing applications to GPUs. The porting strategy can be straightforward. Users should prepare a baseline OpenMP

version running on CPUs. Then the target directive can be inserted around parallel regions. There are only a limited set of accelerator directives and clauses in OpenMP 4.0 to improve parallelism, scheduling, and data reuse, among others. So a simple process is to incrementally apply them one by one to see the effect, guided by performance analysis tools.

However, real applications pose unique challenges to productively apply directive-based programming models. 1) A real scientific application often has complex data types which may not be supported by the language specifications. This can be very problematic since the data types may be extensively used in the code. A common workaround is to manually copy a portion of the complex data object into a variable of a simpler, supported type. It should be possible to design a directive or clause to aid this process and vastly improve the productivity. 2) An application may have non-perfectly nested loops, which can be a candidate for collapsing after simple transformations. One possible way to improve productivity is to extend the collapse(n) clause to accept a flag, like collapse(n:force), to force collapsing across multiple non-perfectly nested loops when applicable. Compilers could simply move all statements between loops into the innermost loop body to form a perfectly nested loop. Users have to ensure the correctness of the code movements. 3) Large-scale DOE applications usually leverage many third-party libraries to increase productivity. Porting such an application may involve porting the underneath libraries to achieve more coverage. However, the complexity of libraries can be a significant barrier to such a porting effort. 4) In an ideal world, users should be able to simply insert directives into existing codes to port to new platforms. However, non-trivial code restructuring may be needed to expose the right granularity of parallelism. 5) Our attempt to exploit special caches on GPUs generated some interesting results. The cache () extension we propose is intuitive, flexible and compatible with existing directives. Using constant memory for LB resulted in significant performance improvements. On the other hand, using shared memory for CNS does not deliver higher performance in our study. The required shared memory space to stage temporary results in stencil computation will increase proportionally to the GPU thread block size. The intuitive implementation to exploit shared memory can easily lead to a much lower parallelism on GPU execution. Additional analysis and optimization support will be helpful to achieve efficient shared memory utilization in stencil applications running on GPU device.

In the future, we plan to explore several research directions: 1) testing extensions to help port complex data types and non-canonical control structures (e.g. non-perfectly nested loops). 2) using more types of scientific applications, such as adaptive mesh refinement and molecular dynamics, to find improvements to the directive-based programming models, 3) further investigation of ways of exploiting shared memory for better performance in real applications, 4) exploring extensions to intuitively and flexibly express semantics related to managing multiple accelerator devices.

References

1. BoxLib. https://ccse.lbl.gov/BoxLib/
2. China's Tianhe-2 Supercomputer Retains Top Spot on 43rd Edition of the TOP500 List. http://www.top500.org/blog/lists/2014/06/press-release/
3. CUDA Zone – The resource for CUDA developers, http://www.nvidia.com/cuda
4. Nvidia visual profiler. WWW page, https://developer.nvidia.com/nvidia-visual-profiler
5. OpenACC: Directives for Accelerators, http://www.openacc-standard.org/
6. Chen, S., Doolen, G.D.: Lattice Boltzmann method for fluid flows. Annu. Rev. Fluid Mech. 30, 329–364 (1998)
7. Colella, P., Graves, D.T., Keen, N., Ligocki, T.J., Martin, D.F., McCorquodale, P., Modiano, D., Schwartz, P., Sternberg, T., , Straalen, B.V.: Chombo software package for amr applications - design document. Tech. rep., Lawrence Berkeley National Laboratory (2009), https://seesar.lbl.gov/anag/chombo/ChomboDesign-3.1.pdf
8. Herdman, J., Gaudin, W., McIntosh-Smith, S., Boulton, M., Beckingsale, D., Mallinson, A., Jarvis, S.: Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:. pp. 465–471 (Nov 2012)
9. Hoshino, T., Maruyama, N., Matsuoka, S., Takaki, R.: CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on. pp. 136–143 (May 2013)
10. Khronos OpenCL Working Group: The OpenCL Specification - Version 1.0. Tech. rep., The Khronos Group (2009)
11. Levesque, J.M., Sankaran, R., Grout, R.: Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-petaflops and Beyond. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 15:1–15:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2388996.2389017
12. Liao, C., Yan, Y., de Supinski, B.R., Quinlan, D.J., Chapman, B.: Early Experiences with the OpenMP Accelerator Model. In: OpenMP in the Era of Low Power Devices and Accelerators (IWOMP'13), pp. 84–98. Springer (2013)
13. Olschanowsky, C., Guzik, S.M.J., Loffeld, J., Hittinger, J., Strout, M.M.: A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In: The International Conference for High Performance Computing, Networking, Storage and Analysis (2014)
14. OpenMP Architecture Review Board: The OpenMP API Specification for Parallel Programming. http://www.openmp.org/
15. Quinlan, D.J., et al.: ROSE compiler project. http://www.rosecompiler.org/
16. Unat, D., Cai, X., Baden, S.B.: Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In: Proceedings of the international conference on Supercomputing. pp. 214–224. ICS '11, ACM, New York, NY, USA (2011)
17. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC: First Experiences with Real-world Applications. In: Proceedings of the 18th International Conference on Parallel Processing. pp. 859–870. Euro-Par'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32820-6_85

18. Wienke, S., Terboven, C., Beyer, J., Mller, M.: A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing, Lecture Notes in Computer Science, vol. 8632, pp. 812–823. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-09873-9_68